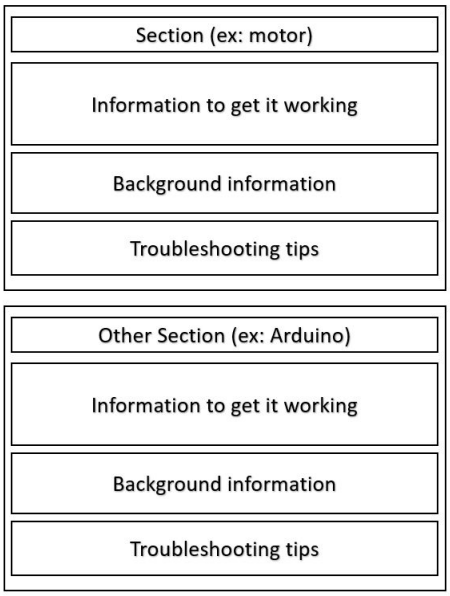
Schulich Space MiniBot Control (V1.0)

By Jordon Cheung and Toshi Taperek

Note: In no way legally or officially to Schulich School of Engineering, the bot electronics that this document describes were created in the Schulich School of Engineering makerspace with the Patron being Chris Simons who at the time of writing an employee of the University of Calgary. Don't sue plz.

Introduction

This is a guide to setup the electronics of a 2019 Makerspace miniBot, this assumes that the physical frame and gear box has been setup before hand and this only talks about the circuits. This guide also includes software that allows for easy wireless controls so that if you are not a person who wants to go through the hassle of figuring out how to program it than this is available to you. This guide is setup in a way to have the most important and least replaceable sections higher up as this document assumes that the miniBots will go through constant changes and thus sections of this guide will become obsolete. Lastly, each section is broken up into 3 parts, the information itself, background and some troubleshooting tips. The information is useful if you want to just get it working and don't care about anything else, background is good if you want some light insight on how the components work and troubleshooting is for when it doesn't work and you want tips on how to fix the problem. It may be useful to read the background information when troubleshooting as it may lend insight on what's actually the problem.



Components

Components that are used:

L298n motor driver	x1
Arduino Nano	x2
nRF24L01	x2
Double throw switch	x2
Potentiometer	x2
DC Motors	x2

Table of Contents

Motor Controller

The Motor driver is one of the fundamental components, it drives the motors. For this you will need the L298n motor driver and 2 DC motors and the arduino nano. Refer to the picture below for the name of the pins that will be used later.

First it’s a good idea to test your DC motors and measure how many amps is the free spin (no load) and when turning the gear box, With this information you can determine the skew that will take place due to imperfections with the motor themselves and the gearbox. This will give you an idea how much one side will be spinning faster than the other, which thus causes bot not to drive straight.

Now Connect one DC Motor to the Motor A terminals and the other to the Motor B terminals. Order doesn't really matter as it’s trivial to change it using a screwdriver or software.

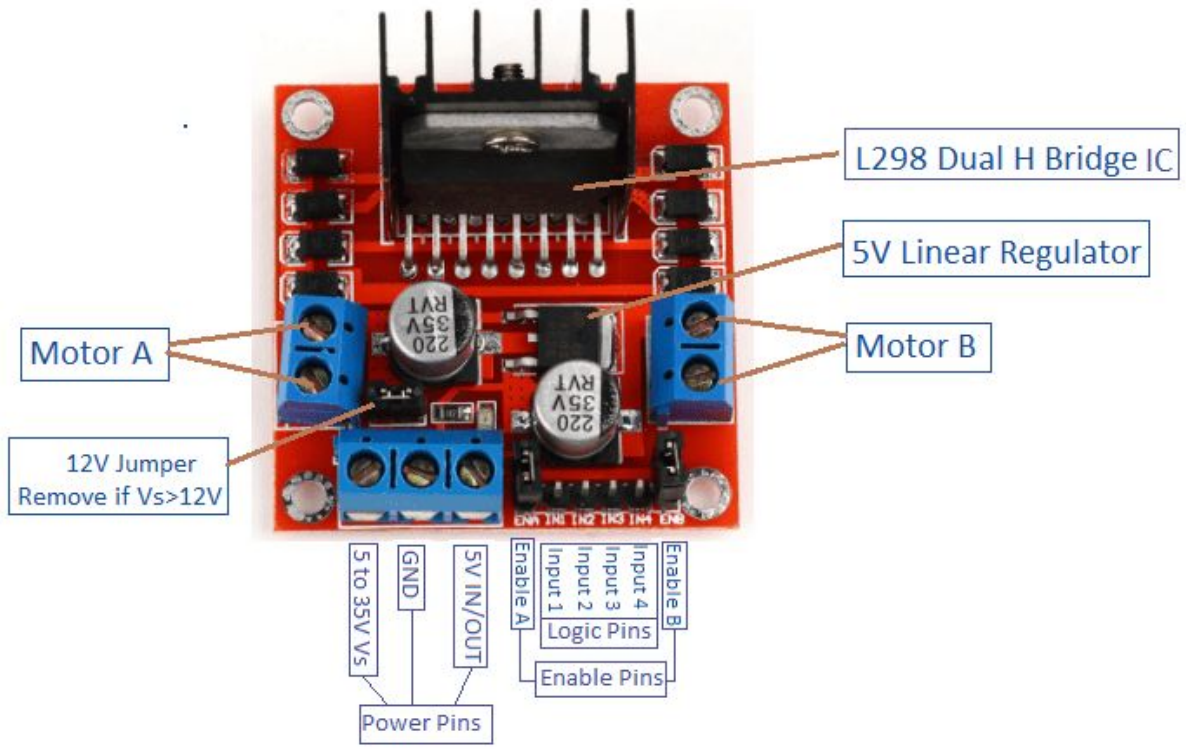
“5 to 35Vs” pin is the power input, you can choose how to power the driver, but it’s recommended that at least 6Vs are supplied as the motors do not spin with any less in testing, additionally more volts means the motors will spin faster giving your bot more speed but also increasing amount of power consumed. Remember to remove the 12Vs jumper if you want to supply greater than 12Vs.

GND goes to ground

5V In/Out should go to the arduino as its a regulated power source and you don’t have to worry about how much you are supplying the drivers.

Remove the jumpers on the Enable A (ENA) and Enable B (ENB) pins. Connect the enable pins closest to the edge of the board to either D3, D5, D6, D9, D10, D11, on the arduino, as these pins have PWM capability (Do not connect enable A and B to the same pin as you want to let them be different), Leave the inner 2 alone.

Inputs 1-4 can all be connected to any (but separate) D pins of the arduino. The two pins for Motor A (inputs 1 and 2) and B (inputs 3 and 4) control which direction each motor spins.



Background

Enables A and B as you might have guessed, enable Motor A or B if they are asserted. The Enable pins are also used to control the speed using PWM (Pulse Width Modulation).

PWM, works by the arduino switching the 5V output pin on and off very quickly. The amount of time it is on or off is what controls the apparent voltage. This is actually much easier to do than producing a true analog output, and will achieve the same effect for most applications. As you can imagine, the longer it remains on means the motor can build up more speed, with maximum speed being when the pin remains on continuously. For example, if the pin is on for 1ms and then off for 1ms, it is on for half of the time and the motor will turn at half speed. To turn it at a quarter speed, the pin would be on for 0.5ms and off for 1.5 ms.

We can use the arduino's PWM functionality with analogWrite(EN, power); where EN is one of the Enable pins and power is a value between 0 and 255. analogWrite only works for D3, D5, D6, D9, D10 and D11 due to the arduino's hardware. Although it may be done manually with another pin using a method called "bit banging", it is very limited and is frowned upon as bad design.

The logic pins are used to control which direction the motors spin. If both are off the motor will not spin, if Input 1 is on, the motor will spin one way, if input 2 is on than it will spin the

other. The program will not allow both pins to be on, as it is an unused state and the behaviour is undefined.

Troubleshooting

You can check if the L298N is on by the red power LED on the board.

Did you tell it to move forward and the bot starts spinning in a circle? Find which wheel is moving in the wrong direction and swap around the wires between the Motor and the Driver, you can also do this by switching the IN1… IN4 around in the code, but this is not recommended as it will cause confusion later on when troubleshooting.

You know the Motor is supposed to be on and all you here is a little whining sound? Try increasing the PWM to higher numbers, it may be that the load on the motor is too great for the current power, you can also increase the voltage of the supply or lubricate the gearbox.

Communication

How to communicate with another chip! For this section you will need 2 Arduino Nanos (one of the nanos may be from the section above) and 2 nRF24L01 wireless modules (nRF), and some wires.

First, we will hook up the nRF to the arduino as shown in the picture below:

MISO to D12

MOSI to D11

SCK to D13

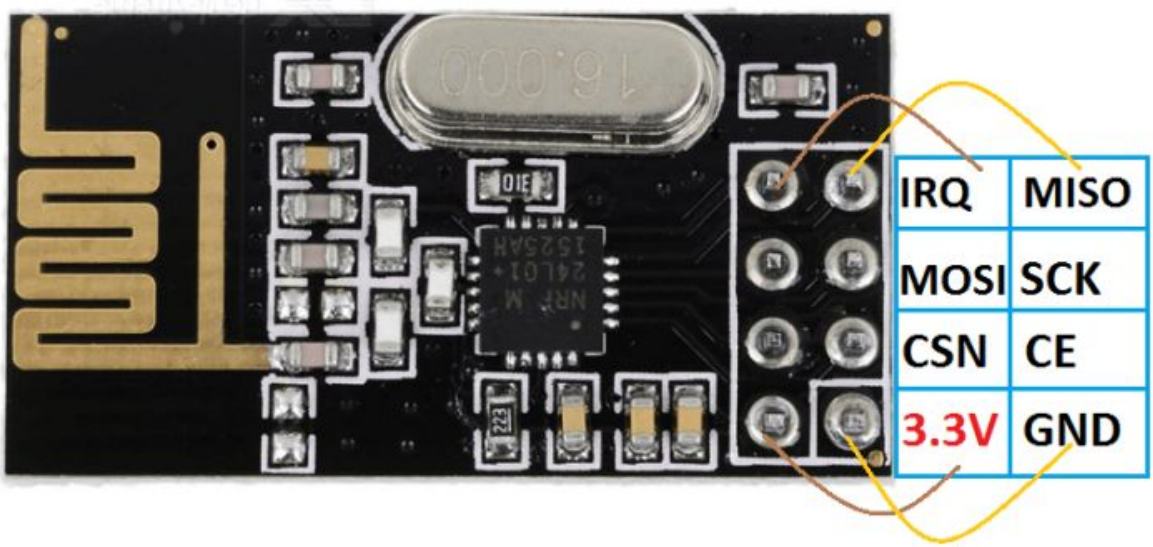
*Note, these pins must be used since the arduino has built-in hardware that only these pins have the functionality to handle.

CSE to any available D pin

CE to any available D pin

3.3v to the arduino’s 3V3 pin

GND to a ground



Repeat this for the other nRF and arduino.

Now it's time for the code. Alongside this document in the zip file should be two ".ino" arduino source code files. The entire code will also be included in Appendix A for completeness. First, let's get the libraries that have been used to make life simpler! In the Arduino IDE, go to "Sketch → Include Libraries → Manage Libraries". Next, In the search bar, type in "RF24". Install the one made by TMRh20 which at this time is on version 1.3.3. Next, find the library called "RF24Network" also by TMRh20, which is version 1.0.9 at this time. Once both are installed, all we need to do now is modify the pins on the top half of the code, example shown below.

```
//CE, CSN can be any pin
#define CE 3
#define CSN 2
//Bluetooth channel
#define CHANNEL 90
//Motors EN pins must be either 3, 5, 6, 9, 10, 11
//Motor Pins
  //MotorA
#define ENA 9
#define IN1 8
#define IN2 7
  //MotorB
#define ENB 6
#define IN3 5
#define IN4 4
```

CE and CSN are going to be whatever D pins you have chosen, and Channel is important to take note of because if multiple transmitters are on the same channel, the receiver may listen to one, the other, or both transmitters. The motor pins will have to be set up first in the section below, but the same follows for those: change the number to the right pin number it is connected with. Lastly, go to "Tools → Processor ATMega328P (old bootloader)" Make sure the board setting in Tools is set to Nano as well.

Background

The nRF module uses SPI which stands for Serial Peripheral Interface. This is one of the most common standard ways for one or more peripherals such as the nRF (called "slaves") to communicate with a microcontroller such as the Arduino (called the "master"). Typically, the slave will define a list of command codes that the master may send in order to do things. Aside from power and ground, SPI uses four wires: MISO, MOSI, SCK, and CS.

MISO stands for “Master In, Slave Out” which is the data that is sent from the Slave, which in this case is the nRF, to the Master, which is the arduino

MOSI stands for “Master Out, Slave In” which is the data that is sent from the Master, which is the arduino, to the Slave, which in this case is the nRF.

SCK (sometimes written CLK) is the Serial ClocK, used to synchronize the master and the slave together. The master will send out a clock pulse whenever it wants to drive the slave to do something such as processing or sending back data to the master. Each clock pulse represents a single action, so to speak, that the slave can carry out. For example, if the slave is supposed to be sending data to the master, the master will send a clock pulse and then read the MISO line, send another, read it again, and so on. The same goes for sending data. The master will set the MOSI line to its value and then send a clock pulse to say “go ahead and read the MOSI line, it’s ready for you” and repeat this several times. Depending on the application, the clock may be driven continuously or intermittently.
Most libraries take care of this for us

CSN (usually called SS for Slave Select, or CS for Chip Select) is what controls which slave the master is talking to. Imagine for a moment if you didn't use a CS and had four slave devices all connected to one arduino each with their own data and clock lines. That would mean that 12 pins on your master will be taken up, but only three would be used at a time. Also, clock signals often require special hardware to reach the desired speeds, so that means extra hardware! Instead, in SPI all slaves share the same SCK, MISO, and MOSI lines, but each slave gets its own CS. The master will send data on the MOSI line to all the slaves, but the CS tells each slave whether or not to listen. SPI chip selects are always active-low, which means high indicates listen and low indicates ignore. If you want to start talking to a certain slave, you first need to switch its chip select from high to low, talk, then set it high again.
These libraries take care of this for us.

CE is the “chip enable” pin, which is an extra fifth pin that the nRFs use to control which mode the nRF is in, such as transmit, receive, or standby. This just allows a quick switching between modes without having to send an entire SPI command first.

The IRQ pin is an interrupt pin. This can be set to go off when data is received and/or transmitted, which can then be used to trigger an *interrupt* on the arduino. This notifies the arduino immediately instead of it having to continuously *poll,* but we will not need to use this feature.

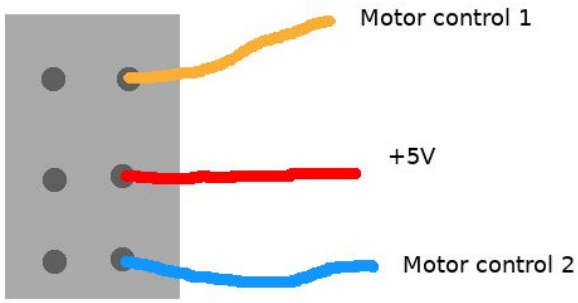
Troubleshooting

If There are no available D pins for CE and CSN then the analog pins can be configured as digital pins as well.

Make sure you have an nRF module rather than an ESP8266, you can tell by the silver bar on the nRF.

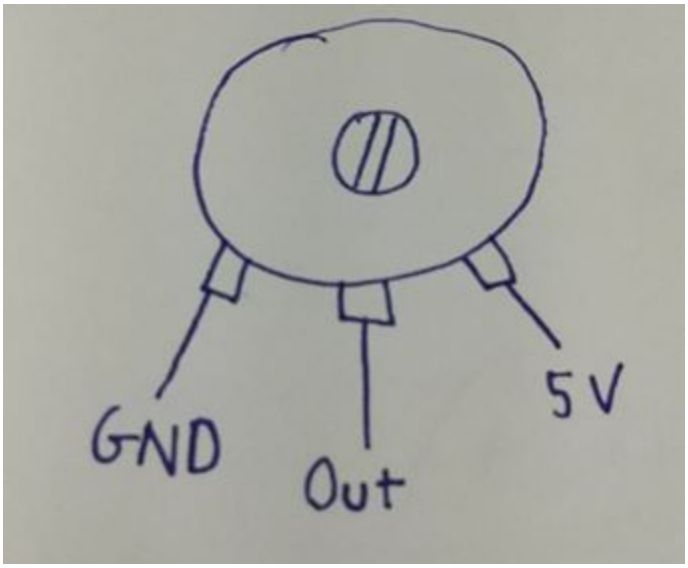
Switch controller

Hopefully by the time you are reading this section this will be obsolete and an app will have been developed and you can just download that. But for this section we'll use 2 double throw switches, 2 potentiometers, 4 resistors and a plastic cap + more wires. This requires a bit of soldering but don't worry, Maker Space offers plenty of opportunities to learn how to do so (or you could just ask someone sweetly to do it for you).



The above diagram assumes you are looking at a double pole double throw switch from the back side. We only need to use one side of these switches so we will leave the other three pins alone. Depending on the position of the switch, either motor control 1, 2 or neither will be powered. These will be the two inputs for one of the motors, so orient your switch so that it connects the correct wire according to the direction of the switch.

The two motor control wires will need to be long enough to reach your arduino, but the +5V wire will only need to meet the respective wire from the other switch and connect to the common power of the controller. We will do this part last.



Next is the Potentiometer. Again, the Out wire will need to be long enough to reach the arduino, but the ground and +5V wires will just meet each other in the middle.

Once both switches and both potentiometers are soldered and in position, bring all the +5V wires together along with another one long enough to reach the arduino, and twist all the stripped ends together. Solder this junction and wrap it in electrical tape or heatshrink if you'd like. Repeat this will all the ground wires. You may want to heatshrink all the long wires together once you're done to make a nicer looking cable to the arduino.

*Note on wire choice, if possible only use red for power (5v) and black for GND, as that is convention and will help other people looking at it know where those wires are supposed to go. Do your best to have as many wire colours other red and black for the outputs as it'll be difficult to tell them apart once they are attached.

*Note on control platform, find and mark areas where you think it would be comfortable and practical to flip the switches (both directions) and turn potentiometers, also note how wide the potentiometer as they might be too wide to fit beside each other.

To connect the controller to the transmitter, connect the outputs of the switches with any open (but different) digital pins, and to each of them, add a resistor with a large resistance (20k Ohms used) between that digital pin and common ground. These are called pull down resistors. Connect the potentiometer outputs to any Analog pins. These will control the speed of each motor. Finally, connect the ground wire and the +5V to +5V on the nano.

*Note: Make sure all the +5V wires are connected to the +5V pin on the arduino not the power from the battery or the power out of the motor driver. This will ensure the signals are at the correct voltage for the arduino to handle.

Background

Pulldown resistors are a way of ensuring that an input is not “floating” what does this mean? Suppose you have a pin connected to nothing (which is the case if you set the switch to neutral) what is the volts running through that pin? 0 right? Not necessarily, voltage is a relative thing, it is the potential difference or difference in the potential electrical energy. Meaning that if it’s connected to nothing that is it’s in reference to? A pull down resistor makes sure that it’s always connected to ground there's always a reference point to compare it to, but as you can imagine letting the current flow directly to ground isn’t a great idea, so the resistor adds, well resistance to the electrons that are flowing.

Potentiometers are variable resistors, meaning that you can change the resistance by turning the knob that is attached to it, we’ll not go how it works physically. Let's say voltage is the amount of energy you have, and let’s say that amount of energy is enough to cross a bridge that is over a river and that there’s a little resting point somewhere along the bridge. Suppose this resting point was in the middle of the bridge, how much energy will you have left? Well if it’s in the middle of the bridge than you will have half your energy left, if it’s closer to the starting side the more energy you will have when you reach it, less if it’s further. This is the same concept for the potentiometer, we are measuring how much voltage is at some point in between the two ends, which in our case is a range from 5v to 0v. The program then translate this into a value from 0-255 for the PWM pin on the receiving end as the power of the motor.

Troubleshooting

If the motors are going in the wrong direction, you can switch around your controller wires or the motor wires. You can test this using the test sequence at the very bottom of the transmitter code

```
Serial.println(dataOut.rightState);
Serial.print("Left State: ");
Serial.println(dataOut.leftState);
Serial.print("Speed Right: ");
Serial.println(dataOut.motorSpeedR);
Serial.print("Speed Left: ");
Serial.println(dataOut.motorSpeedL);
```

They can be activated or deactivated by adding or removing // to the beginning of the line. State 0 is off, 1 is forward and 2 is backwards. Speed is a value between 0-255, 255 being full power.

Lithium Battery:

Although the power source does not matter, we are using a lithium ion battery setup because it's available and it has a ridiculous capacity, for a 2200 mAh at 3.3v out transmitter setup at the lowest power setting can run for 44 hours continuously. This is more difficult than a normal battery setup due to the chemistry of a lithium ion, if it drops below 2.2v than the battery will become unchargeable and essentially useless. Thus we need to add components to prevent this from happening. As this component is not nailed down yet as we had a test run of these components this section will be updated once its been decided what's available.

Expanding Beyond

This section attempts to explain the inner workings of the code, in order to get an understanding of how to modify it to do other things than a simple remote control. First let's look at what data is actually being sent to the receiver

```
//0 = off, 1 = forward, 2 = backward
//Power a value between 0-255
struct motorCommand{
  byte motorSpeedR;
  byte motorSpeedL;
  byte rightState;
  byte leftState;
};
```

This is a user defined structure (a box that contains data) which is called motorCommand what was intended with this is to send which direction the motor direction should be spinning, which was defined (by me) to be 0 for off, 1 for forward and 2 for backwards. You may also easily

define your own states, suppose you wanted to combine both right and left state together, that'll be 9 combinations of various off, forward and backwards. To accomplish this you first change the Transmitter code under the physical control to correctly interpret the inputs it's receiving into your new structure and your new states. Next we move to the Receiver, which you make sure your structure is the same on that side as the two need to know what the other is talking about, then under performAction change it to correctly interpret your new states to do whatever you want to. All the sending and receiving can be left alone as it'll just work with anything you send through it, although it should be noted the data package you're sending has a (testing needed) limit of 24 bytes.

The nrf modules are designed to communicate with multiple other modules, a tutorial can be found in the references to go through this more thoroughly, although this section should be updated once with better information once it has been tested and implemented.

Muhammad Aqib (Motor control && nRF)

https://electronicshobbyists.com/controlling-dc-motors-arduino-arduino-l298n-tutorial/

https://electronicshobbyists.com/nrf24l01-interfacing-with-arduino-wireless-communication/

nRF24L01 data sheet

https://www.sparkfun.com/datasheets/Components/nRF24L01_prelim_prod_spec_1_2.pdf

A guide used to setup a network
https://www.youtube.com/watch?v=xb7psLhKTMA

Appendix A – Code

//THE RECEIVER CODE

```
/**
 * @file Transmitter.cc
 * @author Jordon Cheung
 * https://github.com/jordoncheung99
 * @author Toshi Topeki
 * https://github.com/robotoshi
 *
 */
#include <RF24.h>
#include <RF24Network.h>
#include <SPI.h>
//Due to arduino hardware SCK must be 13, MOSI must be 11 MSIO must be 12
//(Should be the same for all the other communication chip things but this code is for nrf24l01)
//CE, CSN can be any pin
#define CE 3
#define CSN 2
//Bluetooth channel
#define CHANNEL 90
//Motors EN pins must be either 3, 5, 6, 9, 10, 11
//Motor Pins
  //MotorA
#define ENA 9
#define IN1 8
#define IN2 7
  //MotorB
#define ENB 6
#define IN3 5
#define IN4 4



RF24 radio(CE,CSN);
RF24Network network(radio);
const uint16_t thisNode = 01;

//0 = off, 1 = forward, 2 = backward
```

```
struct motorCommand{
  byte motorSpeedR;
  byte motorSpeedL;
  byte rightState;
  byte leftState;
};


void setup() {
  SPI.begin();
  radio.begin();
  network.begin(CHANNEL,thisNode);
  //Just motor things
  pinMode(ENA,OUTPUT);
  pinMode(IN1,OUTPUT);
  pinMode(IN2,OUTPUT);
  pinMode(ENB,OUTPUT);
  pinMode(IN3,OUTPUT);
  pinMode(IN4,OUTPUT);
}

void loop() {
  network.update();
  //Checks if there is any network data
  while(network.available() ){
    RF24NetworkHeader header;
    motorCommand incomingData;
    network.read(header,&incomingData, sizeof(incomingData));
    performAction(incomingData);
  }
}

void performAction(motorCommand data){
  //Left motor Controls
  if(data.leftState == 0){
    leftOff();
  }else if(data.leftState == 1){
    leftForward();
  }else if(data.leftState == 2){
    leftBackward();
  }
  leftPower(data.motorSpeedL);
```

```
    //Right motor Controls
  if(data.rightState == 0){
    rightOff();
  }else if(data.rightState == 1){
    rightForward();
  }else if(data.rightState == 2){
    rightBackward();
  }
  rightPower(data.motorSpeedR);
}



void leftForward(){
  digitalWrite(IN1,HIGH);
  digitalWrite(IN2,LOW);
}

void leftBackward(){
  digitalWrite(IN1,LOW);
  digitalWrite(IN2,HIGH);
}

void leftOff(){
  digitalWrite(IN1,LOW);
  digitalWrite(IN2,LOW);
}

void leftPower(byte power){
  analogWrite(ENA,power);
}

void rightForward(){
  digitalWrite(IN4,HIGH);
  digitalWrite(IN3,LOW);
}

void rightBackward(){
  digitalWrite(IN4,LOW);
  digitalWrite(IN3,HIGH);

}
```

```
void rightOff(){
  digitalWrite(IN4,LOW);
  digitalWrite(IN3,LOW);
}

void rightPower(byte power){
    analogWrite(ENB,power);
}
```

//END OF RECEIVER CODE

//Start of Transmitter Code

```
/**
 * @file Transmitter.cc
 * @author Jordon Cheung
 * https://github.com/jordoncheung99
 * @author Toshi Topeki
 * https://github.com/robotoshi
 *
 */
#include <RF24.h>
#include <RF24Network.h>
#include <SPI.h>
//Due to arduino hardware SCK must be 13, MOSI must be 11 MSIO must be 12
//(Should be the same for all the other communication chip things but this code is for nrf24l01)
//CE, CSN can be any pin
#define PIN_CE  3 // chip enable
#define PIN_CSN 2   // chip select (for SPI)
//Physical Control interface
#define RIGHTUP 9
#define RIGHTDOWN 8
#define LEFTUP 7
#define LEFTDOWN 6
  //Analog Pins
#define RIGHTPOW 0
#define LEFTPOW 1

//0 = off, 1 = forward, 2 = backward
struct motorCommand{
  byte motorSpeedR;
  byte motorSpeedL;
  byte rightState;
  byte leftState;
};

motorCommand dataOut;
uint8_t ch = 0;  // RF channel for frequency hopping

RF24 radio(PIN_CE,PIN_CSN);
RF24Network network(radio);

const uint16_t thisNode = 00;
const uint16_t node01 = 01;
```

```
void setup() {
  Serial.begin(9600);
  pinMode(RIGHTUP,INPUT);
  pinMode(RIGHTDOWN,INPUT);
  pinMode(LEFTUP,INPUT);
  pinMode(LEFTDOWN,INPUT);
  radio.begin();
  network.begin(90,thisNode);
  dataOut = {0,0,1,1};
  dataOut.motorSpeedL = 80;
  dataOut.motorSpeedR = 80;
  dataOut.rightState = 0;
  dataOut.leftState = 0;
}

void loop() {
    network.update();
    RF24NetworkHeader header(node01);
    bool ok = network.write(header, &dataOut, sizeof(dataOut));
    physicalControl();
    delay(100);
}

void physicalControl(){

    //Physical Swtich Control
      //Determines which state the motor should be in
    if(digitalRead(RIGHTUP) == LOW && digitalRead(RIGHTDOWN) == LOW){
      dataOut.rightState = 0;
    }else if(digitalRead(RIGHTUP) == HIGH){
      dataOut.rightState = 1;
    }else if(digitalRead(RIGHTDOWN) == HIGH){
      dataOut.rightState = 2;
    }


    if(digitalRead(LEFTUP) == LOW && digitalRead(LEFTDOWN) == LOW){
      dataOut.leftState = 0;
    }else if(digitalRead(LEFTUP) == HIGH){
      dataOut.leftState = 1;
    }else if(digitalRead(LEFTDOWN) == HIGH){
      dataOut.leftState = 2;
    }
```

```
    //Makes the analog input go from 0-1023, 0 - 255 as PWM has that range.
      //Note might have to make it 0-254 as at 255 PWM is always on which might not work?
    dataOut.motorSpeedR = (int)analogRead(RIGHTPOW)/4;
    dataOut.motorSpeedL = (int)analogRead(LEFTPOW)/4;

    //Serial test code.
    Serial.print("Right State: ");
    Serial.println(dataOut.rightState);
     Serial.print("Left State: ");
    Serial.println(dataOut.leftState);
//     Serial.print("Speed Right: ");
//    Serial.println(dataOut.motorSpeedR);
//     Serial.print("Speed Left: ");
//    Serial.println(dataOut.motorSpeedL);
}
```

//END OF TRANSMITTER CODE